Steps **from Customer's idea** to **a working Frontend**

- A Frontend point of view

v. 2022-10-13. By JV (Will be updated constantly)

This list contains things/steps to study for the Frontend/React/Ajax exam. Some are business case related.

Rough list of things happening parallel or preceding to front-end programming:

1. Customer has the idea
2. Requirements engineering (Interviews, questionnaires, workshops, old system views and outputs, …)
3. UX design, traditionally iterating incrementally between
    a. Use case (Business Process/Navigation) design
    b. View / layout design
    c. (Conceptual Data model design, that was handled in the other doc)
    d. (Logical etc. level Data model design, that was handled in the other doc)
4. UI Mockups. Sometime even partially working Prototypes or demos, sometimes just static Wireframe images

---

Setting up the Frontend project - Node, Nodemon, npm/yarn, React, AJAX with Axios/Fetch, (style libs and files)

5. installing *create-react-app* OR *vite* and running it to generate a running basic node & react frontend development-time project framework (thus no need to run e.g. npm init) with several tools configured, like babel, nodemon.
6. using *npm install xyz* to add more module (react-router-dom, material-ui, icons, form lib?…) references to the *package.json*. Just by one developer who adds them.
7. using *npm install* to install all new modules mentioned in *package.json* after git pull, if you are another developer.
8. the *create-react-app / vite* project has the static parts of the app in *public* folder, e.g. HTML layout, images
9. To be able to use a common theme for all of your Material-UI React components, wrap the root React component inside a *MuiThemeProvider* or similar (Mui = Material-UI) that is given your theme file reference

    Note that the create-react-app / vite generated a framework, which is only the **development-time version**:

    - There **is** Node.js server (for starting an auto-refreshing, by nodemon? webpack?) at port 3000, or other you specify with env variable PORT.
    - Programmers can see a lot of readable files well-organized in several easy-to-browse folders

    The runnable **production version** will be generated with command **npm build**. The runnable version will:

    - …**not** have any Node.js or nodemon at all!!! = no development time server running
    - no .ts/ nor even .jsx files anymore
    - All code will be optimized and packed to only **few** static JavaScript/HTML/CSS files and served from a static folder of your production server to user's browser's JavaScript engine/environment. That environment runs those like there was no development version ever!

10.Testing and running the full-stack project – <u>Windows</u> enlightening us about how/whether the code&views work

i. **Browser window** running the app and showing the HTML based on the Views and Components
ii. **Browser console** (errors, warnings, and our own debugging with console.log or console.dir)
iii. Browser dev tools, **networking window** (Press the Split console to get two browser dev tools visible)
iv. **Front-end** Node **console** (where you run e.g. npm start. Compilation errors etc. in here)
v. **Back-end** Node **console** (where you run e.g. npm start, npm run db:init. Backend prints and errors here)
vi. Console where you created the SSH pipe. Usually you don't have to keep that console open, but with some consoles the SSH (pipe) process dies when you close the console!
vii. **PostMan/VS Code REST client** to check input and output of the needed back-end services
viii. Browser dev tools, e.g. the tool for checking how page will scale for a mobile device
ix. React dev tools windows
x. Browser debugger?
xi. Possible extra Front and back consoles for finding out which processes are running and which TCP/UDP ports they listen to
xii. Backend log files (Linux command tail exists to monitor constantly e.g. last 10 rows in a changing file, in a console window)

Understanding the Front-end part of our architecture and technology

React

React DOM (Virtual DOM)

- React will update just its own virtual DOM, and not the real browser DOM, until it knows all the changes. Thus, e.g. if you Mike was removed and Larry was added, React is able to optimize, not remove nor add any nodes from the browser, but just manipulate browser DOM "Mike"=>"Larry" (The so called 'delta', **change**), if they by luck would be in the same location in the list. (Not best example, but works with any such update that is optimizable)

Root component

- Includes other components, which in turn will be rendered to the page too.
- Mother components often pass data and event-handler function references to its child components.

Your components

- Presentational components, like e.g. CategoryRow,
    - get data in their props from the mother
    - get possibly also the event-handler function references from the mother, map their events to those event-handlers
- Container components, like e.g. CategoryListView, CategoryDetailsView
    - get data via AJAX directly, after some asynchronous await

Think of a mother component that has a child component included inside:

The ways **mother** can pass anything **to the child**:

- **data via props,**
- **methods via props**
- **(via shared context with the useContext hook!)**
- cause a change to happen at backend and the child component could be also connected to backend via some AJAX operations

The ways **child** can pass anything **to the mother**:

- state? – No, child doesn't see mother's state
- props? – No, child doesn't see mother's props
- **by calling the offered functions provided by Mother and adding the data to the function calls as parameter(s).**
- **(via shared context with the useContext hook!)**
- (cause a change to happen at the backend and mother component could be also connected to backend via AJAX (and Mother would somehow know to refresh it's data?))

(((Plus extra: The react-router-dom version 6 allows to create also **routing contexts** where the views belonging to the same nested subsection of routing can share layouts and data **context**! So that would be like the useContext context, but now maybe even better, organized and shared by the nested routing block.)))

React **Component State**

- We can also use states (state fractions, created with the useState hook) of the React Component
- While using the React Component's state we cannot modify it directly by assignment. We should do it with the useState( ) hook created setters (the function setXyz( ) or so) and pass it an ad hoc -object where the only the to-be-changed properties are, the so called 'delta', the changes.

React Component Lifecycle event-handlers (Simple version to remember)

- When component created to eventually come visible to the screen
  - Component JavaScript **object created**. Initial **states** set up. **Hooks** set up. When we have navigated to that view.
  - **return (<rendered components here…>)** - After this the component will come visible
  - **useEffect( )** will be called once after the component mounted to the SPA DOM, e.g. when we have navigated to that view.
- When Components state or props change
  - **return (<rendered components here…>)** - After each state or props change.
  - **useEffect( )** will be called again after states or props change, unless we provide the empty dependency list [ ] to useEffect saying that this useEffect hook is not active then.

10. **((In frontend programming related to foreign key handling**, there are basically three different kind of situations
    a. independent object/table that has an id like Category – No foreign key saved to table
    b. table with an id and foreign key(s) to row(s) in (m)other tables, like Idea has categoryId.
    c. table without an id, but with (possible as the composite primary key) foreign key combo to other tables, like IdeaMember (one member's participation in one idea's work) of a Member for an Idea. It has/needs memberId **and** ideaId.
    Comment would be similar, one member commenting one idea. Foreign keys memberId and ideaId. Though this time primary key is surrogate key id, generated to give identity to each comment.))
11. ((Possibly some UI features might suggest transactional backend operation, where one user action causes many lines to be updated in the database in a single transaction, or rolled back. Thus, frontend will ask for the operation but then in AJAX be ready for both success and failure responses.))

SOME SPECIFIC STUFF TO REMEMBER IN React Development

1. Your component names must start with Caps, small letters reserved for HTML

```
class PlayerListItem extends React.Component { .......
```

// If you would accidentally use small letter, the JSX parser will not know how to work correctly

2. While returning JSX "XML-like" markup from the render method, either wrap the "XML-like" JSX in parentheses ( ), **or** start it from the very same line as the keyword "return". Safer to use the parenthesis.

3. How to read this JSX? `<SomeElement attribute={{a:foo}} />`

The outer { } takes us from the JSX "XML-like" mode to JavaScript mode. Then the inner {a:foo} would just mean creating an ad-hoc JavaScript object with property called 'a' with its value copied from variable 'foo'.

3.b Like above, also in `<SomeElement attribute={{a}} />` we would create a JavaScript object, this time with {a} = object with one property 'a' with its value copied from a variable called 'a' in the current context.

4. In many/some places you'll **have to** use the arrow function in call-back function definition. (As arrow function takes the 'this' reference from the outer lexical context = from the React component). A normal function would **not** bind the "this" to the outer context = component, like we want to happen.

5. The create-react-app sets up the development environment, with other Node.js server, (totally different from our backend Node.js server), possibly trying to listen to port 3000. (To avoid probs, we could use e.g. port 8686 for the development time frontend Node.js "server". Backend Node.js server could use e.g. 8787)

6. The final front-end we get when we first run the **npm build** and then copy the contents of the **/build folder** and publish it on some real web server as static .js .html and .css files E.g. on Proto, Myy, AWS, or Heroku. Then we won't have front-end Node.js server at all anymore! Even just static web server is enough. Most likely with port 80. Then the execution environment will then be just each user's browser's JavaScript environment.

7. Back-end REST API server (E.g. AWS, Heroku, CSC, or Proto) will continue to run Node.js., e.g. with that 8787 port. Myy as static content web server wouldn't be fine for Backend, as Myy doesn't allow us run any servers. ((Though nowadays Myy doesn't allow much anyway, so we don't use it))

8. In HTML we did really call functions in event-handler code:

e.g. <**b**utton on**c**lick="foo(bar);" >xyz</**b**utton>.

Now in React JSX we give code=function to be called:

e.g. <**M**yButton on**C**lick={ ()=>{foo(bar)} } >xyz</**M**yButton>